# CHAPTER 1
# Introduction

The traditional method to construct a computer program when faced with a task to automate or a problem to solve is to write it at source code level. The intended user of the program rarely writes it, since this task requires expert knowledge. Instead, the task to be automated is communicated to a hired programmer, who tries to transform the demands and desires of the user to the code of the program. This program development approach is inherently expensive since there is a large gap, both in theory and in practice, between the original task to be automated as perceived by the intended user and the resulting program as constructed by the programmer.

## 1.1 Purpose

Clearly, it is desirable to reduce the gap between the original task and the program constructed for its automation, to save effort, time and money during the development phase. This calls for more abstract, natural ways to construct computer programs than to program at the code level. Ideally, the end user should be able to communicate the task directly to the computer, removing the need for a hired programmer.

We have chosen the domain of constraint satisfaction problems for the investigation of an alternative approach to constructing source code programs. We discuss the idea of formulating a logic metatheory for a problem class, a theory which can be specialized into different object level theories to represent particular problems by user instantiation of metavariables. Particular instances of the theory, constituting a class of object level theories, are related to potentially efficient programs which can be generated automatically.

The outlined approach is a conception of a program development methodology with the following advantages.

- A compact, declarative problem representation in the well understood formalism of predicate logic. The transparency of the representation facilitates understanding of its fundamental concepts.
- Flexibility—the representation can be specialized, changed and extended at an abstraction level accessible to end users, enabling the formalization of different tasks and problems.
- Interfaces to the representation are written independently, following only the specifications of a metatheory. The design of an interface follows naturally from an intuitive interpretation of the declarative problem representation.
- Efficient computation is achieved by studying the relationship between the problem representation and a suitable computational paradigm, with the objective of transforming from the former to the latter.

To sum up, the present work can be studied from two different perspectives. Most obviously, the thesis is about a high-level representation for stating and solving a class of constraint satisfaction problems. Alternatively, the material of the thesis can be studied as an example of a suggested program development methodology, where the concept of user specialization of a theory is a central aspect. Different tasks and problems are specified as logic formulas and solved by programs which are less natural as problem representations but computationally more efficient.

## 1.2 Background

Logic programming systems with constraint handling capabilities have emerged recently as a powerful tool for stating and solving a wide range of constraint satisfaction problems (CSPs). The main motivation for developing constraint handling logic programming systems, such as CHIP [Hentenryck 1989], is that many constraint satisfaction problems, although elegantly stated and theoretically solvable, cannot be solved in practice (i.e. within reasonable time) by conventional logic programming, Prolog, systems (cf. [Sterling & Shapiro 1986]). In particular, many discrete combinatorial problems in operations research and artificial intelligence can be solved efficiently by constraint logic programming systems with finite domains. We will use the abbreviation CLP(FD) for a constraint logic programming system with finite domain variables (not to be confused with the specific language clp(FD) [Diaz & Codognet 1993]).

From a practical programming point of view, we can say that the difference between a constraint handling logic programming language and ordinary Prolog is that the former is used primarily to represent and solve a certain *class* of problems (CSPs), whereas the latter is a general-purpose language. This does not necessarily mean that the former language is more restricted than the latter, but there exists a motivation for using it only if the problem at hand belongs to the class of CSPs.

From a programming methodology point of view, it is interesting that CLP(FD) is aimed at a certain class of CSPs. It gives us the idea that it may be possible to construct a logic theory in which each problem of this class can be formulated in a more abstract, natural way than writing a special-purpose constraint logic program. When the relationship between problems formulated in the theory and constraint logic programs has been established, computationally efficient programs can be generated automatically. Explicit programming is replaced by high-level programming specification (Figure 1.1).

$$\begin{tikzcd}
\begin{array}{l}\text{CSP theory}\\ \text{instance}\end{array} \arrow[r] & \begin{array}{l}\text{CLP(FD)}\\ \text{program}\end{array}
\end{tikzcd}$$

Figure 1.1 A problem specified in the CSP theory corresponds to a constraint logic program.

A major subject of our investigation is the relationship between a logic theory for abstract specification of discrete CSPs and computationally efficient programs. We indicate a natural relationship between the former and the latter, providing the groundwork for automatic program generation.

The logic theory is a first-order metatheory called the Assignment Theory (*AT*), representing CSPs as constrained resource assignment problems. The methodology for formulating problems in *AT* is a main issue, the goal being to provide the means for building interfaces in which end users themselves can formulate CSPs, as different object level theories (corresponding to efficient CLP(FD) programs). The main parts of the thesis are summarized by Figure 1.2 below.

$$\begin{tikzcd}\text{User interface} \arrow[r, leftrightarrow] & \text{Assignment Theory} \arrow[r, leftrightarrow] & \text{CLP(FD) program}\end{tikzcd}$$

Figure 1.2 The objects of study and the relationships between them.

Our work concerns the study of a logic metatheory for representing a class of problems, the instantiation of specific problems in the theory as object level theories, efficient programs for solving such problems and the relationships between these entities. The concepts of metalogic and metaprogramming are very natural as methodological tools for this work. Metalogic is a well-established concept from philosophy, concerning the discussion of a language (object level) in another language (metalevel), cf. [Tarski 1944]. Recently, metaprogramming has evolved as a suitable formalism for practical realization of such discussions. In particular, metaprogramming and logic programming are used for three different, though interrelated, application areas: *knowledge representation*, *knowledge assimilation* and *knowledge processing*, see [Kowalski 1990]. These three areas correspond to the main subjects of this thesis: knowledge representation in the Assignment Theory, knowledge assimilation by user instantiation of metavariables in the theory and, finally, knowledge processing of the instantiated object theory in order to achieve efficient (CLP(FD)) programs.

## 1.3 Outline

We begin with a discussion of the class of constraint satisfaction problems the proposed Assignment Theory is designed to formalize. This is followed by an abstract description of an assignment system (Chapter 2).

The Assignment Theory is the central, *knowledge representation* part of the thesis. Therefore, we discuss this theory as an independent object of study, postponing the issues of how specific problems are to be instantiated in the theory (knowledge assimilation) and efficient computation of solutions (knowledge processing). We identify the components of the problem class and define the meaning of a solution to a specific problem instance. An Assignment Theory is given by defining general axioms and schemata where metavariables are used to represent the components of infinitely many specific problems (Chapter 3); one problem for each instantiation of the set of schemata. We then define and exemplify constraints which can be expressed in schemata (Chapter 4).

In Chapter 5, a prototype interface is presented, as the extension of an intuitive interpretation of the Assignment Theory.

After the description of the knowledge representation part and the informal understanding of it, we turn our attention to *knowledge assimilation*, which in our case means instantiation of metavariables, specializing *AT* into an object level theory that represents a particular problem. Chapter 6 outlines a formalization of the relationship between our first-order theory containing metavariables on one hand and a user interface on the other.

The *knowledge processing* enterprise of the thesis concerns the problem of achieving efficient programs from different instantiations of *AT*. First, we show the relationship to ordinary Prolog programs (Chapter 7). Then, we take up CLP(FD) as a more efficient problem-solving alternative (Chapter 8) and in Chapter 9 the relation between the Assignment Theory and CLP(FD) is discussed. Optimization of solutions is briefly touched upon in Chapter 10. Some aspects of a flexible environment for obtaining solutions interactively are treated in Chapter 11.

Relations to other work are given in Chapter 12, while conclusions and plans for the future are to be found in Chapter 13. After the bibliography there is an appendix containing additional examples of discrete constraint satisfaction problems, in different forms.

## 1.4 Notation and Typographical Conventions

Predicate calculus is used as the formal language for building the Assignment Theory. In predicate calculus expressions initial lower case letters will be used for predicate symbols and upper case for variables. Constants and functions are written with an initial lower case letter. Predicate calculus and Horn clause expressions are written in `typewriter` font. The single letters `A`, `B` and `C` are reserved to denote metavariables, standing for terms or formulas. Greek letters are used as restricted metavariables, standing for ground terms, or list structures of ground terms. Standard set notation is used and an *ordered* set of two elements is written `[e1, e2]`, where `e1` is the first element and `e2` the last. We use the special symbol `|` as a list constructor; in the expression `[e1, e2] = [U| W]`, `U` is $e_1$ and `W` is `[e2]`.

In predicate calculus and Horn clause expressions we use the following symbols, with their standard meaning.

$\in$, $\notin$, $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, $\neg$, $\forall$, $\exists$, &, $\vee$, $\rightarrow$, $\leftarrow$, $\leftrightarrow$

The symbols are assigned increasing ranks, in the order listed, which means that $\leftrightarrow$ reaches further than &, etc. If necessary, the scope of a symbol is limited by the use of parentheses.

The formula below is an axiom schema of the Assignment Theory, containing the metavariable A.

```
∀X (assignment_objects(X) ↔
       X = A)
```

The symbol A is part of the metalanguage, while ∀, ↔, (, ), =, `assignment_objects` and `X` are symbols of the object language. In order to avoid the mixing of languages, we follow [Kleene 1968] and say that ↔, e.g., appears as a name for itself in sentences containing metalanguage expressions. That is, the logical symbol ↔ is being used *autonymously* [Carnap 1934].

# CHAPTER 2
# Finite Constraint Satisfaction Problems

A finite constraint satisfaction problem (CSP) can be defined as follows. There is a finite set of variables $\{X_1,\ldots, X_n\}$ with related finite domains $D_1,\ldots, D_n$. There is a set of constraints $c(X_{i1},\ldots, X_{ik})$ on the values that can be assigned to the variables. A solution is an assignment of values to the variables from their respective domains, satisfying all constraints.

The formulation above states the CSP framework in a general way. It does not, however, define how problems are to be represented and solved in a computer. Hentenryck and Dincbas observed [Hentenryck & Dincbas 1987] that logic programming languages, such as Prolog, are appropriate tools for describing CSPs, while being inefficient for solving them. They proceeded to suggest an extended logic programming language, CHIP, in which general constraint satisfaction problem solving techniques, such as *forward checking* are incorporated (the CHIP language is described in detail in [Hentenryck 1989]).

## 2.1 A Declarative and Transparent Representation

We essay to elevate the abstraction level when representing CSPs, compared to programming in a constraint logic programming language such as CHIP. We define a predicate calculus representation, intuitively capturing a class of CSPs, where problems to be solved are formulated by instantiating a set of schemata. The representation is transparent, so that a user can understand the notion of constraints and other problem components. Under the assumption that we can design a suitable interface between this representation and the end user, non-trivial problems can be communicated to the computer without programming skill.

No programming constructs or problem solving techniques are present in our predicate calculus representation. We postpone the issue of efficient computation of solutions to Chapter 9.

To summarize, we will suggest a high-level representation in which a class of CSPs can be expressed naturally. The solution to a given problem is found in the appropriate combination of a finite set of assignment objects with a finite set of resources. In general, we will refer to this problem class as *assignment problems* and to our representation as an Assignment Theory ($AT$). We regard our representation as a basic metatheory for the class of constraint satisfaction problems under consideration. The theory will be studied and discussed as a foundation for both natural user interaction and efficient problem solving (computation).

We endeavour to design a system in which a typical end user, unskilled in programming, can specify and solve specific instances of a class of (non-trivial) problems. If this goal is to be reached, the user must understand the fundamental concepts and ideas behind the representation. An axiomatic predicate calculus formulation provides a declarative and transparent representation, with a clear semantics (the advantages of predicate calculus as a knowledge representation formalism have been eloquently put forward elsewhere, cf. [Hayes 1977]). The function of the system can thus be explained naturally. No control information about how the computer should proceed to solve the problem is present in this representation. Moreover, the actual process of problem instantiation can also be defined quite independently of the representation itself, which will be discussed in Chapter 6. Let us be satisfied for the moment with the notion that problem instantiation will be controlled by a metaprogram which inspects the gradually specified theory, enabling the use of advanced interfaces and early detection of impossible specifications.

## 2.2 An Assignment System

We are searching for a representation in which the primitives and solution conditions of a constraint satisfaction problem can be stated as facts and rules in predicate calculus. The Assignment Theory consists of a set of general axioms, *ATg*, and a set of problem specific axioms, *ATs* (in addition to a set of logical axiom schemata and inference rules of a standard first-order predicate calculus formulation). The *ATs* set contains metavariables, which will be replaced by ground terms and formulas when a specific problem is instantiated. The instantiation of *ATs* is performed by a metalevel, giving the object theory *ATg U ATs'*, where *ATs'* is the object theory which results from instantiating the metavariables of *ATs*. That is, each schema is instantiated exactly once for a given problem. A solution to the problem is then logically implied by *ATg U ATs'*, if it follows from the given problem components. So, we can (in theory, at least) make the following deduction both for checking and for generating solutions.

*ATg U ATs'* ⊢ ∃ Y `solution(Y)`

The axioms of *ATg* and *ATs'* are given in full predicate calculus. There is presently no known efficient computational method to perform the outlined deduction. The most straightforward approach to solving this problem is to transform all axioms of the theory into Horn clauses, computing solutions with a Prolog system. Transformations can be made informally, i.e. by writing a program based on the specification or by more formal deductions (see, e.g., [Clark & Tärnlund 1977], [Hansson & Tärnlund 1979]). In Chapter 7 we present a Horn clause program for checking and generating solutions. An informal metalevel view will establish the relationship between a problem instance, represented by *ATs'*, and a Prolog program. This provides the potential for automatic code generation, by formalization of the metaview in a logic program.

The problem with the Prolog approach is that the solution generating program will not solve a CSP particularly efficiently. It will display a typical *generate-and-test* behaviour, which is not a computationally efficient problem solving paradigm for constraint satisfaction problems. This leads us to explore an alternative approach, which is based on another metalevel view of the problem formulation.

Constraint satisfaction problems with finite domains can be coded and solved efficiently in CLP(FD), as shown in [Hentenryck 1989]. Our Assignment Theory is capable of capturing a class of CSPs as object level theories and each such particular problem is related to a CLP(FD) formulation. We will discuss the relationship to some extent, laying the ground for code-generating metaprograms. The results of a CLP(FD) computation can be given to the end user in the original specification format, in a suitable interface. This is important in a flexible environment which permits the user to change suggested solutions, define partial solutions, ask the system for new suggestions or even change the problem representation (and thereby also the corresponding program).

# CHAPTER 3
# An Assignment Theory

In this and the following chapter we characterize the Assignment Theory, i.e. the proposed representation for formalizing a class of discrete constraint satisfaction problems. A solution to a problem will be defined as a set of correct resource assignments to a set of assignment objects. Assignment conditions define the structure of a solution which is also correct if, and only if, it satisfies a set of constraints between the resource assignments.

We will define an example problem and see how its components are represented in *AT*, building the theory by defining axioms.

## 3.1 An Example Problem

The problem is to assign hospital staff, physicians and nurses, to working periods. The working periods are of different kinds: day, night and standby. The physicians are divided into categories according to competence—senior physicians and assistant physicians. Time is divided into Friday, Saturday and Sunday, days and nights and specific hours. In addition, there may exist a variety of constraints concerning staff, time and working periods. The following are examples of constraints: a certain physician is unavailable on Friday morning, some working periods demand a senior physician, a specific nurse and a certain doctor cannot work together, or all physicians should work approximately the same amount of hours.

## 3.2 Assignment Objects

Let us represent the working periods as a set of assignment objects. Assignment objects can represent such diverse things as technical components or hours of a working schema. The assignment objects of our example problem are day, night and standby working periods. They are naturally depicted as a hierarchy, where wp (short for working period) is a main class and day, night, etc., are further specializations of this class.

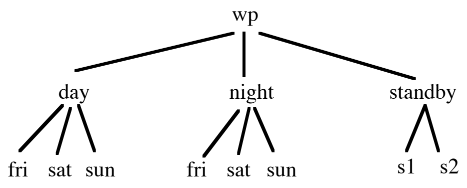

Figure 3.1 A hierarchical description of assignment objects.

We can represent an assignment object as a structure where the first element is the main class, `wp`, the second is either `day`, `night` or `standby`, etc. The hierarchy of Figure 3.1 corresponds to eight such structures, from `[wp, day, fri]` to `[wp, standby, s2]`. The set of assignment objects is represented in *AT* by instantiating the metavariable A of **Schema1**.

**Schema1**
```
∀X (assignment_objects(X) ↔
    X = A)
```

where A is a metavariable which is to be replaced by a set of unique assignment objects, X, where each object is a constant or a list structure. In the latter case, the elements of the list structure are nodes in an assignment object hierarchy. This schema, and the ones to follow, can be instantiated only once for a given problem. That is, if the problem formalization is changed, for example by adding an assignment object, a new schema instantiation will replace the previous one.

In the working period example, the set of assignment objects, `X`, is:

```
{[wp, day, fri], [wp, day, sat], [wp, day, sun],
 [wp, night, fri], [wp, night, sat], [wp, night, sun],
 [wp, standby, s1], [wp, standby, s2]}.
```

Each member of a set of assignment objects is an assignment object. This is easily defined by a general axiom.

```
∀U (assignment_object(U) ↔
    ∃X (assignment_objects(X) & U ∈ X))
```

The reason for representing each assignment object above as a list structure, instead of a simple constant as `wpdayfri`, is that the structure provides the means for natural reference to different sets of assignment objects. This, in turn, provides the means for natural constraint formulations. For instance, we can refer to all night or day periods in a constraint, without having to enumerate them explicitly. Let us, for the sake of the argument, show a definition of a night period.

```
∀U (night_period(U) ↔
    ∃V (U = [wp, night, V] & assignment_object(U)))
```

The outlined technique will be a basis for expressing constraints on solutions, where reference is made to sets of assignment objects and sets of resources.

## 3.3 Resources

In our example there are two kinds of resources: hospital *staff* and *time*, as depicted below.

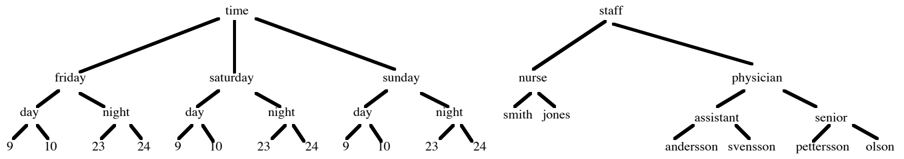

Figure 3.2 Resource hierarchies.

A resource is of the same structure as an assignment object. The set of all resources consists of all structures reflecting paths in the hierarchies—from `[time, friday, day, 9]` to `[staff, physician, senior, olson]` in Figure 3.2.

More generally, a set of resources is an instance of the metavariable `B` of an axiom schema.

**Schema2**

```
∀X (resources(X) ↔
    X = B)
```

where `B` is to be replaced by a set of resources, `X`, where each resource is a constant or a list structure. In the latter case, the elements of the list structure are nodes of a resource hierarchy.

The set of resources in our example domain is as follows.

```
{[time, friday, day, 9], [time, friday, day, 10],
 [time, friday, night, 23], [time, friday, night, 24],
 [time, saturday, day, 9], [time, saturday, day, 10],
 [time, saturday, night, 23], [time, saturday, night, 24],
 [time, sunday, day, 9], [time, sunday, day, 10],
 [time, sunday, night, 23], [time, sunday, night, 24],
 [staff, nurse, smith], [staff, nurse, jones],
 [staff, physician, assistant, andersson],
 [staff, physician, assistant, svensson],
 [staff, physician, senior, pettersson],
 [staff, physician, senior, olson]}
```

An individual resource, `R`, is defined as a member of a resource set.

```
∀R (resource(R) ↔
    ∃X (resources(X) & R ∈ X))
```

Analogously to the reference to sets of assignment objects, we can use variable parts of expressions to refer to sets of resources. For instance, we can define the midnight hours of the resource set.

```
∀R (midnight_hour_resource(R) ↔
    ∃V1∃V2 (R = [time, V1, V2, 24] & resource(R)))
```

When a variable is referenced only once in a logic formula, as is the case for both V1 and V2 above, we sometimes use the underscore, _, to stand for the variable and omit explicit quantification. It will be clear from context if _ stands for a universally or existentially quantified variable. We will also use the convention of letting just ∃ (or ∀) represent the existential (or universal) closure of variables not within the scope of other quantifiers.

## 3.4 Resource Assignment Specification

A correct combination of assignment objects and resources will constitute a *solution* to a given problem. This is formalized by a general axiom.

```
∀Y (solution(Y) ↔
    assignments(Y) & correct(Y))
```

There are two demands on the solution; it must consist of assignments and it has to be correct. The first demand expresses the basic conditions of an assignment problem, as exemplified below, where time and staff are assigned to different working periods. Specific times are set for all working periods, while the assignments of staff resources are more loosely specified, with the aid of existential variables.

```
∀Y (assignments(Y) ↔
∃ (Y = [[[wp, day, fri], [A1, A2, A3, A4]],
        [[wp, day, sat], [A5, A6, A7, A8]],
        [[wp, day, sun], [A9, A10, A11, A12]],
        [[wp, night, fri], [A13, A14, A15, A16]],
        [[wp, night, sat], [A17, A18, A19, A20]],
        [[wp, night, sun], [A21, A22, A23, A24]],
        [[wp, standby, s1], [A25, A26, A27, A28, A29, A30, A31]],
        [[wp, standby, s2], [A32, A33, A34, A35, A36, A37, A38]]] &
A1 = [time, friday, day, 9] & A2 = [time, friday, day, 10] &
A3 = [staff, physician, assistant, V1] & A4 = [staff, nurse, V2] &
A5 = [time, saturday, day, 9] & A6 = [time, saturday, day, 10] &
A7 = [staff, physician, assistant, V3] & A8 = [staff, nurse, V4] &
A9 = [time, sunday, day, 9] & A10 = [time, sunday, day, 10] &
A11 = [staff, physician, assistant, V5] &
A12 = [staff, nurse, V6] & A13 = [time, friday, night, 23] &
A14 = [time, friday, night, 24] &
A15 = [staff, physician, V7, V8] & A16 = [staff, nurse, V9] &
A17 = [time, saturday, night, 23] &
A18 = [time, saturday, night, 24] &
A19 = [staff, physician, V10, V11] & A20 = [staff, nurse, V12] &
A21 = [time, sunday, night, 23] & A22 = [time, sunday, night, 24] &
A23 = [staff, physician, V13, V14] & A24 = [staff, nurse, V15] &
A25 = [time, friday, day, 9] & A26 = [time, friday, day, 10] &
A27 = [time, friday, night, 23] & A28 = [time, friday, night, 24] &
A29 = [time, saturday, day, 9] & A30 = [time, saturday, day, 10] &
A31 = [staff, physician, V16, V17] &
A32 = [time, saturday, night, 23] &
A33 = [time, saturday, night, 24] & A34 = [time, sunday, day, 9] &
A35 = [time, sunday, day, 10] & A36 = [time, sunday, night, 23] &
A37 = [time, sunday, night, 24] &
A38 = [staff, physician, V18, V19] &
resource(A1) & ... & resource(A38)))
```

It is stated that the day working periods must have an assistant physician and a nurse assigned, while a night working period is to be combined with a physician (assistant or senior) and a nurse. The first standby period covers the first half of the weekend, while the second takes the other half, each with one physician assigned. More generally, the solution is defined to be an ordered structure of eight assignment objects, each with an ordered list of assignments. Each assignment is a resource and if its specification contains variables, more than one resource may exist as a potential instantiation.

The example of assignment specifications above is an instantiation of the following schema.

**Schema3**

$\forall$Y (assignments(Y) $\leftrightarrow$
    $\exists$ (Y = [ [$\Delta_1$, [$A_1$, ..., $A_r$]], ..., [$\Delta_n$, [$A_s$, ..., $A_t$]] ] &
        $A_1$ = $E_1$ & ... & $A_t$ = $E_t$ &
        resource($A_1$) & ... & resource($A_t$)))

where Y is an ordered structure, each $\Delta_i$ is a variable free assignment object structure, belonging to the set specified in **Schema1** and each $E_i$ is either a resource specification, which may contain variables, or a variable identical to $A_i$.

The level of detail of the Assignment Theory may seem too low for natural problem specification. There is no reason, however, why a user of the system should be as detailed as the theory itself when instantiating particular problems. For instance, a user should not be expected to explicitly write the existential variables in the resource assignment expression above. A user interface to the theory is naturally seen as the realization of an intended interpretation of the theory. If *AT* is employed to represent problems where concrete objects are to be combined, the theory is interpreted accordingly, resulting in an interface where the assignment objects are naturally reflected, e.g. as graphical objects on a screen. In a work assignment domain, the interpretation of assignment objects would probably be different, resulting in other presentations. Similarly, there are many ways to interpret the constraints of the next section. It will be the task of an interface designer to interpret *AT* and construct an interface according to the interpretation. The fundamental concepts of the theory will thus be communicated mainly through the user interface. An example interface, resulting from an interpretation of *AT*, is discussed in Chapter 5.

## 3.5 Summary

We have defined the first parts of the Assignment Theory; assignment objects, resources and the basic specification of their potential combinations. A solution is a combination of resource assignments which, in addition, must be correct. Solution correctness will be defined in terms of relational and numerical constraints, the subjects of the next chapter.

# CHAPTER 4
# Formulation of Constraints

In the previous chapter we described the formalization of assignment objects, resources and the general conditions for combining them in a solution. Now, we will define the means for expressing constraints on a solution. That is, a solution must also be correct, as defined below.

```
∀Y (solution(Y) ↔
    assignments(Y) & correct(Y))

∀Y (correct(Y) ↔
    satisfies_relational_constraints(Y) &
    satisfies_numerical_constraints(Y))
```

A solution is correct if, and only if, all relational and numerical constraints on the solution are satisfied. We will give schemata for expressing these categories of constraints. Relational constraints concern how resources must, or must not, be combined with assignment objects, sometimes depending on other resource assignments. Numerical constraints are used to state limitations on the usage of resources in a solution.

## 4.1 Relational Constraints

We will introduce relational constraints by describing how some restrictions from the example hospital domain are formalized in *AT*. This is meant to give an understanding of the different kinds of constraints which can be expressed, as well as introducing their logical form.

Let us assume that we want to state the following constraint in our theory: "if senior physician Olson is assigned to a night working period, he must work with nurse Jones". In order to express this formally, we map the constituents of the natural language statement into the concepts of *AT*, using appropriate predicate calculus symbols. A first order formula follows.

```
∀Y (satisfies_relational_constraints(Y) ↔
    ∀([U1, Z1] ∈ Y & U1 = [wp, night, _] &
       RSpec1 ∈ Z1 & RSpec1 = [staff, physician, senior, olson] →
       RSpec2 ∈ Z1 & RSpec2 = [staff, nurse, jones]))
```

As soon as Olson is assigned to a night period, Jones must work with him—otherwise the axiom is not satisfied. Naturally, the constraint is satisfied if Olson is *not* assigned to a night working period, since we then have a false antecedent of a material implication.

In another example, we assume that: "a night working period must always have either a senior physician or Svensson assigned". This exemplifies the need for especially qualified physicians during some periods. Again, the formalization comes naturally (the constraint is stated within the *satisfies relational constraints* axiom above).

```
∀ ([U1, Z1] ∈ Y & U1 = [wp, night, _] →
   ∃ (RSpec1 ∈ Z1 &
      (RSpec1 = [staff, physician, senior, _] v
       RSpec1 = [staff, physician, assistant, svensson])))
```

Next, we want to state that a certain combination of resources is *not* allowed in a solution, exemplified by the statement: "nurse Jones refuses to work together with senior physician Olson on a night working period”.

```
¬∃ ([U1, Z1] ∈ Y & U1 = [wp, night, _] &
    RSpec1 ∈ Z1 & RSpec2 ∈ Z1 &
    RSpec1 = [staff, physician, senior, olson] &
    RSpec2 = [staff, nurse, jones])
```

The first three example constraints all treat relationships only between resources of the same assignment object. There are no relations between resources of different assignment objects. We will now confront our theory with some examples where such relations do occur. Two constraints are presented, followed by their formalization. (i) “if Andersson is assigned to a working period, then a senior physician must be assigned to a simultaneous standby period” (ii) “if a night working period is scheduled to the same time as a standby period, then at least one of the periods must have a senior physician assigned”.

```
∀ ([U1, Z1] ∈ Y &
    RSpec1 ∈ Z1 & RSpec2 ∈ Z1 &
    RSpec1 = [time, R1, R2, R3] &
    RSpec2 = [staff, physician, assistant, andersson] →
    ∃ ([U2, Z2] ∈ Y & U1 ≠ U2 & U2 = [wp, standby, _] &
       RSpec3 ∈ Z2 & RSpec4 ∈ Z2 &
       RSpec3 = [time, R1, R2, R3] &
       RSpec4 = [staff, physician, senior, _])) &
∀ ([U1, Z1] ∈ Y & U1 = [wp, night, _] &
   [U2, Z2] ∈ Y & U2 = [wp, standby, _] &
   RSpec1 ∈ Z1 & RSpec2 ∈ Z2 &
   RSpec1 = [time, R1, R2, R3] &
   RSpec2 = [time, R1, R2, R3] →
   ∃ (RSpec3 ∈ Z1 & RSpec3 = [staff, physician, senior, _] v
      RSpec3 ∈ Z2 & RSpec3 = [staff, physician, senior, _]))
```

The main novelty in these constraints is that they concern resources of two separate assignment objects. A typical relationship between resources of different assignment units is that they must be equal, as expressed by the two occurrences of the specification `[time, R1, R2, R3]` in the first constraint above.

By using negation we can express that some relationship is forbidden. Let us formalize the constraint: “it is forbidden to have two different senior physicians assigned to different working periods at the same time in the night”.

```
¬∃ ([U1, Z1] ∈ Y &
    [U2, Z2] ∈ Y & U1 ≠ U2 &
    RSpec1 ∈ Z1 & RSpec2 ∈ Z1 & RSpec3 ∈ Z2 & RSpec4 ∈ Z2 &
    RSpec1 = [time, R1, night, R2] &
    RSpec2 = [staff, physician, senior, R3] &
    RSpec3 = [time, R1, night, R2] &
    RSpec4 = [staff, physician, senior, R4] & R3 ≠ R4)
```

In some real-world constraints one refers to the order of resources. For instance, if we want to state that no physician should work more than two nights in sequence, we must be able to tell that Saturday comes after Friday, etc. We will therefore incorporate the means for ordering resource hierarchies and for expressing distances between resources.

## 4.2 Distances between Resources

Let us assume that the distance between Friday and Saturday is one day, that there is one hour between nine and ten of Friday morning, etc., as illustrated in the time hierarchy of Figure 4.1.

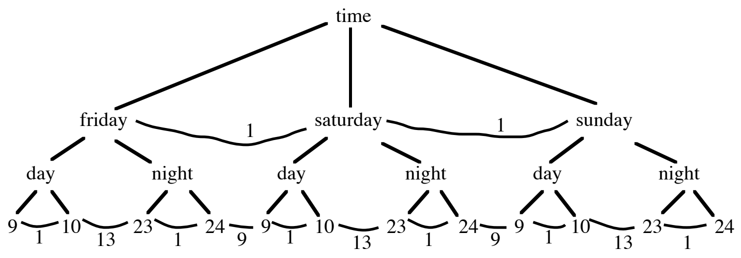

Figure 4.1 Distances in the time resource hierarchy.

Naturally, it is now up to us to interpret the number one between Friday and Saturday to mean one *day*, while the same number between nine and ten means one *hour*. This is important for the use of asserted distances in constraint formulations. Also, in order to be meaningful, distances should be specified only between elements at the same level of a resource hierarchy.

Let us now see how we can formalize distances in the theory. If we look at the illustration above, we see that the distance one between nine and ten o'clock on Friday is easily asserted by specifying this distance between two resources—`[time,` `friday, day, 9]` and `[time, friday, day, 10]`. The distance between Friday and Saturday is different—we need to define the distance between the resource classes `[time, friday]` and `[time, saturday]`. The set of distances between resources and resource classes for a given problem is asserted by instantiating the fourth axiom schema of our theory.

**Schema4**

```
∀ (resource_distance(R1, R2, D) ↔
   R1 = B11 & R2 = B12 & D = ρ1 v ... v
   R1 = Bn1 & R2 = Bn2 & D = ρn)
```

where each $B_{ij}$ and $B_{ik}$ are resources, or resource classes, at the same level in the resource hierarchy and each $\rho_i$ is an integer.

```
∀ (resource_distance(R1, R2, D) ↔
   R1 = [staff, physician, assistant] &
   R2 = [staff, physician, senior] & D = 1 v
   R1 = [time, friday] & R2 = [time, saturday] & D = 1 v
   R1 = [time, saturday] & R2 = [time, sunday] & D = 1 v
   R1 = [time, friday, day, 9] &
   R2 = [time, friday, day, 10] & D = 1 v
   R1 = [time, friday, day, 10] &
   R2 = [time, friday, night, 23] & D = 13 v
   R1 = [time, friday, night, 23] &
   R2 = [time, friday, night, 24] & D = 1 v
   R1 = [time, friday, night, 24] &
   R2 = [time, saturday, day, 9] & D = 9 v
   R1 = [time, saturday, day, 9] &
   R2 = [time, saturday, day, 10] & D = 1 v
   R1 = [time, saturday, day, 10] &
   R2 = [time, saturday, night, 23] & D = 13 v
   R1 = [time, saturday, night, 23] &
   R2 = [time, saturday,  night, 24] & D = 1 v
   R1 = [time, saturday, night, 24] &
   R2 = [time, sunday, day, 9] & D = 9 v
   R1 = [time, sunday, day, 9] &
   R2 = [time, sunday, day, 10] & D = 1 v
   R1 = [time, sunday, day, 10] &
   R2 = [time, sunday, night, 23] & D = 13 v
   R1 = [time, sunday, night, 23] &
   R2 = [time, sunday, night, 24] & D = 1)
```

We define a general axiom which states the distance between two resources, or resource classes (which are assumed to be at the same level of the resource hierarchy).

```
∀ (distance(R1, R2, D) ↔
   resource_distance(R1, R2, D) v
   ∃ (resource_distance(R1, R3, D1) &
      distance(R3, R2, D2) &
      D = D1 + D2))
```

Next, we define some axioms which enable us to conveniently state that a resource or a resource class is before or after some other resource or class of resources.

```
∀ (before(S1, S2) ↔
   ∃ (distance(S1, S2, D) & D > 0))

∀ (before_equal(S1, S2) ↔
   ∃ (distance(S1, S2, D) & D ≥ 0))

∀ (after(S1, S2) ↔
   ∃ (distance(S2, S1, D) & D > 0))

∀ (after_equal(S1, S2) ↔
   ∃ (distance(S2, S1, D) & D ≥ 0))
```

Given the general axioms above and suitable instantiations of resource distances, it is possible to write constraints which talk about demanded or forbidden distances between resources. An example constraint is “nurse Jones must work two night periods in sequence”, as formalized below.

```
∃ ([[wp, night, U1], Z1]∈ Y &
   [[wp, night, U2], Z2]∈ Y & U1 ≠ U2 &
   Z1 = [RSpec1, RSpec2, RSpec3, RSpec4] &
   Z2 = [RSpec5, RSpec6, RSpec7, RSpec8] &
   RSpec4 = [staff, nurse, jones] &
   RSpec8 = [staff, nurse, jones] &
   RSpec2 = [time, R1, _, _] & RSpec5 = [time, R2, _, _] &
   distance([time, R1], [time, R2], 1))
```

This concludes our exemplification of the formulation of relational constraints in *AT*. A metalevel schema for representing relational constraints is stated below.

**Schema5**
```
∀Y (satisfies_relational_constraints(Y) ↔
    A1(Y) & ... & An(Y))
```

where each $A_i$ is an implication

$\forall(B(Y) \rightarrow \exists C(Y))$

or a disjunction of *m* implications

$(\forall(B(Y)_1 \rightarrow \exists C(Y)_1) \vee \ldots \vee \forall(B(Y)_m \rightarrow \exists C(Y)_m))$

where B and C are built from the predicates: $\in$, $\notin$, $=$, $\neq$, $>$, $\geq$, $<$, $\leq$, `distance`, `before`, `before_equal`, `after` and `after_equal` and the arithmetic symbols `+`, `-` and `*`, using the logical connectives `&` and `v`. Negation, $\neg$, is used only to negate the consequent part. Variables in the antecedent are universally quantified over the whole constraint, while variables introduced in the consequent are existentially quantified. Arithmetic expressions can be stated over integer constants and any quantified variable representing an integer, using `+`, `-` and `*`. Parentheses are used to limit the scope of symbols such as `v`, when necessary. The antecedent of a constraint may be empty, in which case there are only existential variables, if any, in the expression.

In addition to the relational constraints, we distinguish a class of constraints which is used to express numerical demands on resource employment in a solution.

## 4.3 Constraints on Resource Usage

The last schema of *ATs* is used to express constraints on resource usage. In this schema we can state that certain resources, as defined by a specification, must occur a specified number of times in a solution, allotted to certain assignment objects.

**Schema6**
```
∀Y (satisfies_numerical_constraints(Y) ↔
    A1(Y) & ... & An(Y))
```

where each $A_i$ is an expression of the form

```
∃ (number_of_combinations(Ei, Δi, Y, P) &
   ρj ≤ P ≤ ρk))))
```

where $E_j$ is to represent a resource specification and $\Delta_i$ is an assignment object specification. Resource and object specifications may contain existentially quantified variables. The metavariables $\rho_j$ and $\rho_k$ will represent numbers, $\rho_j \leq \rho_k$, denoting an interval to which the number of resource combinations, P, must belong.

Below we give an example instantiation, specifying that (i) “physician Olson must work two or three night periods” (ii) “the maximum total use of all senior physicians is 15 periods”.

```
∀Y (satisfies_numerical_constraints(Y) ↔
    ∃ (number_of_combinations([staff, physician, senior, olson],
                              [wp, night, U1], Y, P) &
      2 ≤ P ≤ 3) &
    ∃ (number_of_combinations([staff, physician, senior, R1],
                              U1, Y, P) &
      0 ≤ P ≤ 15))
```

The `number_of_combinations` predicate is a relation between a resource specification, an assignment object specification, the solution and a number. The intuition is that all assignment units of Y with an assignment object equal to the specification of the constraint, and a resource which is equal to the resource specification, make up the number of combinations.

## 4.4 Summary

We have defined two kinds of constraints—relational and numerical—which are intended to naturally formalize some typical components of CSPs. In particular, a single relational constraint can be used to state that a relationship involving many individual objects of different classes is demanded or forbidden. Such constraints need not be changed if only the sets of individual objects, e.g. staff resources, change over time. The formalization of constraints is intended to be as natural as possible with respect to the real-world problem statements, but the abstraction level will have to be further elevated in a useful assignment system. The subject of a user interface to the problem representation of *AT* is treated in the next chapter.

# CHAPTER 5
# The Interface between User and Theory

The purpose of the Assignment Theory is to provide a high-level representation in which a class of CSPs can be formulated naturally, i.e. in a form which is close to the original problem statements, as perceived by the user of the assignment system in the real world. Still, the *AT* formalism is too cumbersome for the end user, who cannot be expected to be skilled in logic. The question is therefore if we can find a natural mapping from the entities of *AT*—assignment objects, constraints, etc.—to an intuitive user interface. We claim that this is the case. In fact, when formulating *AT* in the first place, our starting point was an intuitive, informal understanding of the problem domain—assignment objects as a hierarchy, natural language formulations of relational constraints, etc. This first conceptualization of the objects and relationships of the domain did not contain any technical constructions of logic, such as universal or existential quantification of variables. The line we will follow is therefore to regard the user interface as an intended interpretation of the theory, where the technicalities of the logic formalism, for instance explicit quantification of variables, are presented more abstractly yet precisely.

We will outline a prototype interface implementation between *AT* and the end user. It should be emphasized that this interface is merely a sketch and that fundamentally different versions could well be imagined and designed. The interface should be seen as a modular part of an assignment system, communicating with *AT* only through a formal relationship, which will be described in the next chapter. The objective of our prototype is to see if we can find an intuitive interpretation of *AT* that maps naturally into a comprehensible user interface.

## 5.1 A Prototype Interface

Let us assume that the different parts of *ATs*, i.e. the six schemata, are specialized by user interaction in the following order: assignment objects, resources, resource distances, resource assignments, relational constraints and numerical constraints. At the outset the interface will, in some format, have access to the general part of the Assignment Theory,