The *at-most-one* constraint can be expressed in terms of the following agent.

xcell(1, N, N).
xcell(0, _, _).

Note that this agent is determinate (Section 2.5) if the first argument is known, or if the last two arguments are known and different.

For a sequence of squares V_1 to V_k, we can now express that at most one of these squares is 1 using the xcell agent as follows.

xcell(V_1, N, 1),
xcell(V_2, N, 2),
…,
xcell(V_k, N, k)

If more than one V_i is 1, the variable N will be bound to two different numbers, and the constraint will fail. Let us call this constraint at_most_one(V_1, …,V_k), thus avoiding the overhead of having to write a program to create it.

An at_most_one constraint will clearly only have solutions where at most one square is given the value 1, but note also the following propagation effects. If one of the V_i is given the value 1, its associated xcell agent becomes determinate, and can therefore be reduced. When it is reduced, N is given the value i, and the other xcell agents become determinate, and can be reduced, giving their variables the value 0.

The *at-least-one* constraint can be expressed in terms of the following agent.

ycell(1, _, _).
ycell(0, S, S).

Note that this agent too is determinate if the first argument is known, or if the other two arguments are known and different.

For a sequence of squares V_1 to V_k, we can express that at least one of these squares is 1 using the ycell agent as follows.

S_0 = true,
ycell(V_1, S_0, S_1),
ycell(V_2, S_1, S_2),
…,
ycell(V_k, S_{k-1}, S_k),
S_k = false

If all the squares are 0, a chain of equality constraints, $S_0 = S_1$, $S_1 = S_2$, …, will connect the symbols ‘true’ and ‘false’, and the constraint will fail. This constraint we call at_least_one(V_1, …, V_k).

Again note the propagation effects. If a variable is given the value 0, then its associated ycell agent becomes determinate. When it is reduced, its second and third arguments are unified. If all variables but one are 0, the second argument of the remaining ycell agent will be ‘true’ and its third argument will be ‘false’,

and it will therefore be determinate. When it is reduced, its first argument will be given the value 1.

Thus, not only will these constraints avoid the undesirable cases, but they will also detect cases where information can be propagated. When no agent is determinate, and therefore no information can be propagated, alternative assignments for variables will be explored by trying alternatives for the xcell and ycell agents.

A program solving the *n*-queens problem can now be expressed as follows.

- For each column, row, and diagonal, consisting of a sequence of variables V_1, …, V_k, the constraint at_most_one(V_1, …, V_k) has to be satisfied.
- For each column and row, consisting of a sequence of variables V_1, …, V_n, the constraint at_least_one(V_1, …, V_n) has to be satisfied.
- The composition of these constraints is the program.

Note that when information is propagated, this will affect other agents, making them determinate. This will often lead to new propagation. One such case is illustrated in Figure 3.4.

1	0	0	0
0	0	V_{23}	V_{24}
0	V_{32}	0	V_{34}
0	V_{42}	V_{43}	0

Figure 3.4. A state when solving the 4-queens problem

The above grid represents the board, and in each square is written the variable representing it, or its value if it has one. We will now trace the steps leading to the above state. Initially, all variables are unconstrained, and all the constraints have been created. Let us now assume that the topmost leftmost variable (V_{11}) is given the value 1. It appears in the row V_{11} to V_{14}, in the column V_{11} to V_{41}, and in the diagonal V_{11} to V_{44}. Each of these is governed by an at_most_one constraint. By giving one variable the value 1, the others will be assigned the value 0 by propagation.

A second case of propagation is that in Figure 3.5 (next page), where V_{12} and V_{24} are assumed to contain queens, and propagation of the above kind has taken place.

Here we examine the propagation that this state will lead to. Notice that in row 3, all variables but V_{31} have been given the value 0. This triggers the at_least_one constraint governing this row, giving the last variable the value 1,

which in turn gives the variables in the same row, column, or diagonal (only V_{41}) the value 0. Finally, V_{43} is given the value 1 by reasoning as above.

0	1	0	0
0	0	0	1
V_{31}	0	0	0
V_{41}	0	V_{43}	0

Figure 3.5. Another state when solving the 4-queens problem

In comparison, *n*-queens programs written using *finite domain constraints* do not exploit the fact that both rows and columns should contain exactly one queen (e.g., [Van Hentenryck 1989; Carlson, Haridi, and Janson 1994]). They are, however, much faster since propagation is performed by specialised machinery.

A better solution can be obtained if all the xcell and ycell agents are ordered so that those governing variables closer to the centre of the board come before those governing variables further out. If at some step alternatives have to be tried for an agent, values will be guessed for variables at the centre first. This is a good heuristic for the *n*-queens problem.

3.6 INTEGRATION

So far, the different paradigms have been presented one at a time, and it is quite possibly by no means apparent in what relation they stand to each other. In particular the relational and the constraint satisfaction paradigms have no apparent connection to the process paradigm. Here, this apparent dichotomy will be bridged, by showing how a process-oriented application based on the solver for the *n*-queens problems could be structured.

The basic techniques for interaction with the environment (e.g., files and the user) are discussed first, and then an overall program structure is introduced.

3.6.1 Interoperability

The idea underlying interoperability is that an AKL agent sees itself as living in a world of AKL agents. The user, files, other programs, all are viewed as AKL agents. If they have a state, e.g., file contents, they are closer to objects, such as those discussed in Section 3.3. It is up to the AKL implementation to provide this view, which is inherited from the concurrent logic programming languages.

A program takes as parameter a port to the “operating system” agent, which provides further access to the functionality and resources it controls. An inter-

face to foreign procedures adds glue code that provides the necessary synchronisation, and views of mutable data structures as ports to agents.

The details of this form of interoperability have not yet been worked out. The examples use imaginary, although realistic, primitives, as in the following.

```
main(OS) :=
        send(create_window(W, [xwidth=100, xheight=100]), OS),
        send(draw_text(10, 10, 'Hello, world!'), W).
```

Here it is assumed that the agent main is supplied with the "operating system" port OS when called. It provides window creation, an operation that returns a port to the window agent, which provides text drawing, and so on.

For some kinds of interoperability, a consistent view of don't know nondeterminism can be implemented. For example, a subprogram without internal state, such as a numerical library written in C, does not mind if its agents are copied during the course of a computation. For particular purposes, it is even possible to copy windows and similar "internal" objects. But the real world does not support don't know nondeterminism. It would hardly be possible to copy an agent that models the actual physical file system; nor would it be possible to copy an agent that models communication with another computer.

The only solution is to regard this kind of incompleteness as acceptable, and either let attempts to copy such unwieldy agents induce a run-time error, or give statements a "type" which is checked at compile-time, and which shows whether a statement can possibly employ don't know nondeterminism.

3.6.2 Encapsulation

To avoid unwanted interaction between don't know nondeterministic and process-oriented parts of a program, the nondeterministic part can be *encapsulated* in a statement that hides nondeterminism. Nondeterminism is encapsulated in the guard of a conditional or committed choice and in bagof.

When encapsulated in the guard of a conditional or committed choice, a nondeterministic computation will eventually be pruned. In a conditional choice, the first solution is chosen. In a committed choice, any solution may be chosen.

When encapsulated in bagof, all solutions will be collected in a list.

More flexible forms of encapsulation can be based on the notion of *engines*. An engine is conceptually an AKL interpreter. It is situated in a server process. A client may ask the engine to execute programs, and, depending on the form of engine, it may interact with the engine in almost any way conceivable, inspecting and controlling the resulting computation. A full treatment of engines for AKL is future work.

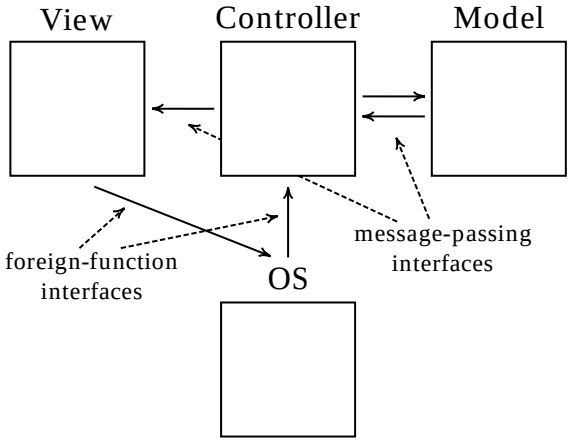

Figure 3.6. The basic program structure

3.6.3 A Program Structure

Responsibilities can be assigned to the components of a program as illustrated by Figure 3.6. The *view* presents output on appropriate devices. The *controller* interprets input and acts as a relay between the model and the view. The *model* is where the actual computation takes place, and this is also where don't know nondeterminism should be encapsulated.

To exemplify the above, we apply it to an *n*-queens application. We assume the existence of a don't know nondeterministic *n*-queens agent

n_queens(N, Q) := … .

which returns different assignments Q to the squares of an N by N chess board. It is easily defined by adding code for creation of constraints for different length sequences, and code for creating sequences of variables corresponding to rows, columns, and diagonals on the chess board. No space will be wasted on this trivial task here. We proceed to the program structure with which to support the application.

```
main(P) :=
        initialise(P, W, E),
        view(V, W),
        controller(E, M, S, V),
        model(M, S).
```

The initialise agent creates a window accepting requests on stream W and delivering events on stream E. The view agent presents whatever it is told to by the controller on the window using stream W. The model delivers solutions on the stream S to the *n*-queens problems submitted on stream M. The controller is driven by the events coming in on E. It submits problems to the model on stream M and receives solutions on stream S. It then sends solutions to the view agent on V for displaying.

Let us here ignore the implementation of the initialise, view, and controller agents. The interesting part is how the don't know nondeterminism is encapsu-

lated in the model agent. We assume that we are satisfied with being able to get either one or all solutions from the particular instance of the *n*-queens problem, or getting the reply that there are no solutions (for N = 2 or N = 3).

```
model([], S) :-
    → S = [].
model([all(N) | M], S) :-
    → bagof(Q, n_queens(N, Q), Sols),
      S = [all(Sols) | S1],
      model(M, S1).
model([one(N) | M], S) :-
    → (       Q : n_queens(N, Q)
          → S = [one(Q) | S1]
      ;       S = [none | S1] ),
      model(M, S1).
```

As described above, don't know nondeterminism is encapsulated in bagof and conditional choice statements.

CHAPTER 4

A COMPUTATION MODEL

Ideally, a computation model should serve a number of purposes:

- It should be a mental model for programmers.
- It should be a useful guide for implementors.
- It should be easy to use for investigating formally the properties of programs and the language.

The AKL computation model serves all of these purposes. It serves well as a mental model for programmers, it is an often useful level of abstraction of the behaviour of an implementation, and it has been used to show results about the language.

The computation model is formalised as a labelled transition system on configurations [Plotkin 1981]. It is defined by straightforward computation rules corresponding to different computation steps, and a structural rule propagating the effect of individual steps to the whole configuration. The major technicality is the introduction and control of don't know nondeterminism.

The model presented here has evolved from earlier models of KAP and AKL [Haridi and Janson 1990; Janson and Haridi 1991; Franzén 1991; Janson and Montelius 1992; Franzén 1994]. The main novelty of the current version is the use of the statement syntax for programs, as opposed to the clausal syntax of previous versions.

4.1 DEFINITIONS AND PROGRAMS

Assume given sets of *variables*, *constraint names*, and *program atom names*. A *constraint atom* is an expression of the form $c(X_1, \ldots, X_n)$, where c is a constraint name and $X_1, \ldots, X_n$ are variables. Similarly, a *program atom* is an expression of the form $p(X_1, \ldots, X_n)$, where p is a program atom name and $X_1, \ldots, X_n$ are different variables. An *atom* is a constraint atom or a program atom. The variables

in an atom are called *parameters*. The number of parameters—determined by *p* or *c*—is called the *arity* of the atom.

The remaining (abstract) syntactic categories pertaining to programs follow.

⟨definition⟩ ::= ⟨head⟩ := ⟨body⟩

⟨head⟩ ::= ⟨program atom⟩

⟨body⟩ ::= ⟨statement⟩

⟨statement⟩ ::= ⟨atom⟩ | ⟨composition⟩ | ⟨hiding⟩ | ⟨choice⟩ | ⟨aggregate⟩

⟨composition⟩ ::= ⟨statement⟩, ⟨statement⟩

⟨hiding⟩ ::= ⟨set of variables⟩ : ⟨statement⟩

⟨choice⟩ ::= ⟨sequence of clauses with the same guard operator⟩

⟨clause⟩ ::= ⟨set of vars⟩ : ⟨statement⟩ ⟨guard operator⟩ ⟨statement⟩

⟨guard operator⟩ ::= '→' | '|' | '?'

⟨aggregate⟩ ::= aggregate(⟨variable⟩, ⟨statement⟩, ⟨variable⟩)

The clauses of a choice statement have the same guard operator. To the guard operators correspond *conditional* choice ('→'), *committed* choice ('|'), and *nondeterminate* choice ('?') statements, respectively. The parameters of a head atom are called *formal parameters.* A variable is *bound* in a statement if all occurrences are in clauses, hiding, or aggregates, with corresponding occurrences in the corresponding hidden sets of variables or the first position of the aggregate. Otherwise, it is *free*. Variables in the body of a definition are either bound or formal parameters . A *program* is a finite set of definitions for different program atom names, satisfying the condition that every program atom name occurring in the program has a definition in the program.

In this chapter, bagof is generalised to the notion of an *aggregate*. The different alternative results for a don't know nondeterministic agent p(X) are grouped as follows. Let $\theta_1, \ldots, \theta_n$ be the results for p(X), in which local variables have been renamed apart, and let $X_1, \ldots, X_n$ be the different renamings of X. Let the operation *unit*(Y) bind Y to the unit of the aggregate (Y = [] in bagof), and let the operation *collect*(X, Y_1, Y) form an aggregate Y of a solution X and another aggregate Y_1 (Y = [X | Y_1] in bagof). Executing the statement

aggregate(X, p(X), Y)

will yield a result of the form

θ_1, *collect*(X_1, Y_1, Y), …, θ_n, *collect*(X_n, Y_n, Y_{n-1}), *unit*(Y_n)

For example,

bagof(X, member(X, [a,b,c]), Y)

yields the result

X_1 = a, Y = [X_1 | Y_1], X_2 = b, Y_1 = [X_2 | Y_2], X_3 = c, Y_2 = [X_3 | Y_3], Y_3 = []

Thus, the aggregate operation is defined by its unit and collect operations. Often, these are constraints, but in the following they may be arbitrary statements. In addition, the aggregate may be regarded as ordered or unordered, the unordered version being preferred if the collect operation is associative and commutative. For example, an unordered aggregate can reliably count the solutions, or add them up if they are numbers. There can also be operational reasons for preferring one or the other.

4.2 CONSTRAINTS

Constraint atoms are regarded as atomic formulas in some constraint language, for simplicity based on first order classical logic. Other logics might be useful, but such generality is not of interest in the present context.

The symbols σ, τ, and θ will be used for conjunctions of constraint atoms, called *constraints*. In the following ∃σ stands for the existential closure of σ, and ∃V for (any permutation of) the quantifier sequence $\exists X_1 \cdots \exists X_n$, where V is a set $\{X_1, \ldots, X_n\}$ of variables. We allow V to be empty, in which case ∃V is mere decoration.

We assume given some constraint theory **TC** defining the following logical properties of constraints:

1. σ is *satisfiable* iff **TC** ⊨ ∃σ
2. σ is *quiet w.r.t.* θ *and* V iff **TC** ⊨ θ ⊃ ∃Vσ, where the variables in V do not occur in θ
3. σ and θ are *incompatible* iff **TC** ⊨ ¬(σ ∧ θ)

A constraint which is not quiet is called *noisy*.

The symbols **true** and **false** denote variable-free atomic formulas for which it holds that **TC** ⊨ **true** and **TC** ⊨ ¬**false**. No further assumptions concerning the properties of **TC** will be made.

For the constraint theory of equalities between *rational trees* [Maher 1988], we can make the following observations. We need only consider constraints of the form **false**, or of the (satisfiable) *substitution* form $v_1 = t_1 \wedge \ldots \wedge v_n = t_n$, where v_i are different variables and t_i are variables not equal to any v_j or constructor expressions of the form $f(u_1, \ldots, u_k)$, where f is a tree constructor of arity k (≥ 0) and u_i are variables, since a constraint can always be reduced to either of these forms by *unification* [Lassez, Maher, and Marriott 1988]. A variable v is *bound* by a substitution if it is the left-hand or right-hand side of an equation. The constraint **true** is regarded as a special case of a substitution, where n is 0. A constraint is satisfiable if it can be reduced to a substitution. A constraint σ is quiet w.r.t. a satisfiable constraint θ in substitution form, and a set of variables V, if the reduced form τ of θ ∧ σ is satisfiable, and all variables bound by τ are either in V, bound to a variable in V, or are bound by θ. Constraints σ and θ are incompatible if θ ∧ σ can be reduced to **false**. See Section 4.4.7 for further discussion of rational tree constraints.

Other constraint theories that we will have reason to mention are those of *finite trees* [Maher 1988], *feature trees* [Aït-Kaci, Podelski, and Smolka 1992], *records* [Smolka and Treinen 1992], and *finite domains* [Van Hentenryck, Saraswat, and Deville 1992].

4.3 GOALS AND CONTEXTS

Goals are expressions built from statements and combinators called boxes. Their syntax is defined as follows.

$$
\begin{aligned}
\langle \text{goal} \rangle &::= \langle \text{global goal} \rangle \mid \langle \text{local goal} \rangle \\
\langle \text{global goal} \rangle &::= \langle \text{or-box} \rangle \mid \langle \text{and-box} \rangle \\
\langle \text{or-box} \rangle &::= \mathbf{or}(\langle \text{sequence of global goals} \rangle) \\
\langle \text{and-box} \rangle &::= \mathbf{and}(\langle \text{sequence of local goals} \rangle)^{\langle \text{constraint} \rangle}_{\langle \text{set of variables} \rangle} \\
\langle \text{local goal} \rangle &::= \langle \text{statement} \rangle \mid \langle \text{choice-box} \rangle \mid \langle \text{aggregate box} \rangle \\
\langle \text{choice-box} \rangle &::= \mathbf{choice}(\langle \text{sequence of guarded goals} \rangle) \\
\langle \text{guarded goal} \rangle &::= \langle \text{global goal} \rangle \, \langle \text{guard operator} \rangle \, \langle \text{statement} \rangle \\
\langle \text{aggregate box} \rangle &::= \mathbf{aggregate}(\langle \text{variable} \rangle, \langle \text{or-box} \rangle, \langle \text{variable} \rangle)
\end{aligned}
$$

The variables in the set of variables associated with an and-box are called the *local variables* of the and-box. The constraint associated with an and-box is called the *local constraint store* of the and-box.

In the following, the letters R, S, and T stand for sequences of goals. Sequences are formed from goals and other sequences using the associative comma operator. (No confusion is expected, even though comma is also used for composition.) Sequences may be empty. The symbol ε stands for the empty sequence, when the need arises to name it explicitly. The letter G stands for a goal, and the letters A and B for statements. The letters *u*, *v*, and *w*, stand for single variables, and the letters U, V, and W stand for sets of variables. Letters are decorated with indices and the like as needed. The symbol '%' stands for a guard operator.

An and-box of the form $\mathbf{and}()^{\sigma}_{V}$ is called *solved* and may be written as σ_V. An or-box of the form **or**() may be written as **fail**.

We will need two more concepts: that of a *context*, which is a goal with a "hole", and that of the *environment* of a context, which are the constraints "visible" from the hole.

We define *contexts* inductively as follows. The symbol λ denotes the hole, and the symbol χ denotes a context.

- λ is a context.
- if χ is a context then $\mathbf{and}(R, \chi, S)^{\sigma}_{V}$, **or**(R, χ, S), **choice**(R, (χ % A), S), and **aggregate**(*u*, χ, *v*) are contexts.